

## Aula 4

### Conceitos básicos de Linguagens Imperativas

- Influenciadas pela arquitectura de computadores de Von Neumann, com abstrações para a memória e o processador: variáveis e comandos.
- Os programas são decompostos em passos de computação correspondendo à sua execução em hardware:
  - Memória: Dados e Programa
  - Unidade de controlo (CPU) e Unidade aritmética (ALU)
  - Contador de programa
  - Ciclo *Fetch-Decode-Execute*
- Os dados são acedidos usando memória virtual (administrada pelo sistema de operação e que permite usar memória massa para memória de execução) mas o programa pode alocar e libertar memória durante a execução.
- Procedimentais

Descreve-se a seguir alguns os conceitos específicos, em especial propriedades de variáveis e de controlo de fluxo entre comandos.

### Variáveis

**Nome ou Identificador** normalmente uma string que é especificada na sintaxe concreta; são definidos nas *declarações de variáveis*. Podem também haver *declarações* de funções, classes ou tipos.

**Tipo** indica quais os valores que podem ser atribuídos e que operações estão definidas. Pode não existir, ser definido estaticamente (compilação) ou dinamicamente (execução). No primeiro caso a verificação de tipos pode também ser estática.

**Endereço** posição de memória associada à variável. Por vezes designado *valor esquerdo*  $\ell$  ou *referência*. A associação de nomes de variáveis a posições de memória não é simples: variáveis diferentes podem ter o mesmo nome e uma posição de memória pode ser referida por várias variáveis (*aliases*).

**Valor** conteúdo da posição de memória. Designado por *valor direito*  $r$  ou *! (desreferenciação)*. As atribuições alteram o valor da variável. As variáveis podem ser *constantes* se o seu valor não puder ser alterado (e apenas definido quando é declarada).

**Duração** Tempo em que uma variável está alocada a uma posição de memória: alocação/desalocação permitem associar/libertar memória a variáveis. A memória é normalmente composta por:

- memória estática: declarações de variáveis conhecidas na compilação e que não mudam (endereço/tamanho) durante a execução (*variáveis estáticas*). Desvantagem: não é alocada dinamicamente e desperdiça recursos.
- pilha de execução: usada na chamada (recursiva) a subprogramas e para parâmetros e variáveis locais (*variáveis dinâmicas de pilha* (frame variables)). Permite alocar variáveis (arrays) de tamanho dependente do input (em blocos) e temporariamente.
- *heap* (listas ligadas de blocos): para variáveis alocadas dinamicamente durante a execução. Não é necessário conhecer o tamanho dum array antes da alocação. Permite que arrays sejam retornados de um subprograma. Podem ser

**Explícitas** São associadas a apontadores e são alocadas/desalocadas durante a execução, por comandos. O espaço alocado depende do tipo da variável. A libertação de memória é feita por um *garbage collector*. Na linguagem C as funções da biblioteca `malloc()` e `free()` permitem manipular a alocação/desalocação de blocos de memória.

**Implícitas** é alocada memória quando são atribuídos valores durante a execução (p.e. Javascript). A deteção de erros é mais difícil e menos eficiente.

**Âmbito** parte do programa na qual as variáveis são visíveis (blocos). Os blocos podem ser subprogramas (procedimentos ou funções), objectos, classes, métodos, etc. Pode ser:

- Estático: o âmbito de cada variável pode ser determinado na compilação. Neste caso é procurada uma declaração da variável do âmbito mais local para o mais geral.
- Dinâmico: depende da sequência de chamadas a subprogramas durante a execução (e.g. Javascript e Python).

## Exemplo

Considera o seguinte programa

```
int x = 2;
void f();
{
  x = 1;
};
int main(int argc, char *argv[])
{
  {
    int x = 0;
    f();
```

```

printf("%d", x);
};
printf("%d", x);
return 0;
}

```

- Se for usado âmbito estático para as variáveis o primeiro **printf** escreve 0 porque a variável  $x$  em  $f()$  é a declarada globalmente e não a local ao bloco. O segundo **printf** escreve 2, porque usa o  $x$  definido globalmente. Este é o método da linguagem C.
- se o âmbito for dinâmico, o primeiro **printf** escreve 1, porque em  $f()$  o  $x$  é o local ao bloco em que é chamado. O segundo **printf** escreve 2, porque usa o  $x$  definido globalmente.

## Atribuição

Uma atribuição

$$< \textit{nome - de - variavel} > := \textit{expressao}$$

guarda na *posição de memória* referida pelo nome da variável o *valor* da expressão. Por exemplo, em  $x := y + 1$  para

- $x$  é identificada a sua referência (ou valor esquerdo)
- $y$  é identificado o seu valor (direito) (desreferenciação)
- Se o nome da variável denota uma referência então a atribuição pode ser denotada por  $x := !y + 1$  para garantir que do lado esquerdo é uma referência e do lado direito uma desreferenciação. Assim numa expressão uma variável  $x$  deve ser usada da forma  $!x$ .
- mas isso não acontece em geral na sintaxe das linguagens de programação (mas sim na sua semântica/compilação).

Em linguagens tipificadas é verificado se os tipos de ambos os lados da atribuição são compatíveis.

## Instruções de Controlo

**Condicionais ou de Seleção** Usados para a escolha de fluxos alternativos de controlo. Podem ser **if-then-else** ou **switch-case(s)**.

**Iterativos (ciclos)** Repetição de um conjunto de instruções até que uma dada condição é verdadeira. O teste pode ser feito antes ou depois da primeira execução das instruções. Podem ser

**Controlados por contador** O número de repetições depende do valor de um contador (em geral, variável inteira).

**Controlados logicamente** As repetições são realizadas até que uma dada condição seja verdadeira.

Pode haver uma instrução que permita a saída do ciclo (e.g. **break**), mas isto torna os programas mais difíceis de compreender e analisar.

## Programas Estruturados: Blocos e Subprogramas

**Bloco** Conjunto de instruções delimitadas por separadores (e.g. { e }) que podem conter declarações (de constantes, variáveis ou tipos) e comandos.

**Subprograma** Bloco que tem um nome e pode ser chamado. Quando chamado, suspende a execução de quem o chamou, o bloco é executado e quando termina, o controlo passa para quem o chamou. E.g: procedimentos em Pascal ou Fortran; funções em C e métodos em Java.

- Quando declarados, para além do nome podem ter uma *lista de parâmetros* ou argumentos (formais)
- Quando chamados os argumentos têm valores (actuais)
- Há vários mecanismos de *passagem de parâmetros*:
  - call-by-value (por valor)** o valor do parâmetro actual é copiado para o formal
  - call-by-name (por nome)** uma cópia do parâmetro actual é passada para o subprograma
  - call-by-reference (por referência)** um apontador para o parâmetro actual é passado para o subprograma.
- Podem ter variáveis locais (declaradas ou não) e aceder a variáveis globais.
- Subprogramas podem definir e chamar outros subprogramas.
- Em geral, sempre que um subprograma é chamado é criado um *registo de activação* na pilha de memória, que contém:
  - variáveis locais
  - parâmetros
  - apontador para o (sub)programa que o chamou
  - endereço de retorno (da instrução onde foi chamado)
  - informação sobre variáveis locais

O acesso aos vários tipos de variáveis depende do âmbito (estático ou dinâmico) que é implementado.

## Blocos e Declaração de Variáveis

Vamos aumentar a linguagem **While** para estes constructores, a que chamamos a linguagem **Block**.

### Novas Categorias sintácticas e comandos

**Dec<sub>V</sub>** Declarações de variáveis,  $D_V$

$$\begin{aligned} c &::= x := a \mid \text{skip} \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \\ &\quad \mid \text{while } b \text{ do } c \mid \text{begin } D_V \text{ c end} \\ D_V &::= \text{var } x := a; D_V \mid \varepsilon \end{aligned}$$

$D_V$  é uma string (ou lista) de declarações onde  $\varepsilon$  é a string vazia. Então temos que  $D_V; \varepsilon \equiv D_V$

**Exemplo 4.1.** Para derivarmos  $\text{var } x := 1$ ; pelo não terminal  $D_V$  temos que considerar a primeira alternativa  $\text{var } x := a$ ;  $D_V$  e depois considerar que  $D_V$  deriva  $\varepsilon$ .

As variáveis declaradas num bloco são locais a esse bloco.

**Exemplo 4.2.** No programa seguinte determinar o valor de  $x$  e  $y$  no final.

```
begin var y := 1;
  x := 1;
  begin var x := 2;
    y := x + 1
  end;
  x := y + x
end
```

O valor final de  $x$  é 4 e o de  $y$  é 3.

### Semântica operacional para comandos do Block

$$\text{block}_{sn} \frac{\langle D_V, s \rangle \longrightarrow_D s' \quad \langle c, s' \rangle \longrightarrow s''}{\langle \text{begin } D_V \text{ c end}, s \rangle \longrightarrow s'' [DV(D_V) \mapsto s]}$$

onde  $DV$  determina as variáveis declaradas em  $D_V$ :

$$\begin{aligned} DV(\text{var } x := a; D_V) &= \{x\} \cup DV(D_V) \\ DV(\varepsilon) &= \emptyset \end{aligned}$$

e  $s'[X \mapsto s] : Var \rightarrow \mathbb{Z}$  é definido por

$$s'[X \mapsto s](x) = \begin{cases} s(x) & \text{se } x \in X \\ s'(x) & \text{se } x \notin X \end{cases}$$

Isto permite modelar variáveis locais, uma vez que quando o bloco termina o valor das variáveis é restaurado para o valor que tinham previamente.

### Semântica operacional para as declarações de variáveis

$D_V$  corresponde a uma sequência de atribuições e portanto irá alterar o estado (memória)  $s$ .

O sistema de transições  $\longrightarrow_D$  é definido por

$$\begin{array}{l} \text{var}_{sn} \quad \frac{\langle D_V, s[x \mapsto \mathcal{A}[[a]] s] \rangle \longrightarrow_D s'}{\langle \text{var } x := a; D_V, s \rangle \longrightarrow_D s'} \\ \text{empty}_{sn} \quad \langle \varepsilon, s \rangle \longrightarrow_D s \end{array}$$

Sucessivamente as variáveis declaradas na lista  $D_V$  assumem os valores que lhe são atribuídos.

**Exercício 4.1.** Usando a semântica natural para blocos prova os valores final da variável  $x$  no Exemplo 4.2:

```
begin var y := 1;
  x := 1;
  begin var x := 2;
    y := x + 1
  end;
  x := y + x
end
```

◇

Sejam

$c_1 = x := 1; (c_2; x := y + x)$   
 $c_2 = \text{begin var } x := 2; y := x + 1 \text{ end.}$

Então

$$\frac{\frac{\langle \varepsilon, s[y \mapsto 1] \rangle \longrightarrow_D s_0}{\langle \text{var } y := 1, s \rangle \longrightarrow_D s_0} \quad \frac{T_1}{\langle c_1, s_0 \rangle \longrightarrow s_5}}{\langle \text{begin var } y := 1; c_1 \text{ end}, s \rangle \longrightarrow s_6}$$

$T_1 :$

$$\frac{\langle x := 1, s_0 \rangle \longrightarrow s_1 \quad \frac{\frac{T_2}{\langle c_2, s_1 \rangle \longrightarrow s_2} \quad \langle x := y + x, s_2 \rangle \longrightarrow s_5}{\langle c_2; x := y + x, s_1 \rangle \longrightarrow s_5}}{\langle x := 1; (c_2; x := y + x), s_0 \rangle \longrightarrow s_5}$$

$T_2 :$

$$\frac{\frac{\langle \varepsilon, s_1[x \mapsto 2] \rangle \longrightarrow_D s_3}{\langle \text{var } x := 2, s_1 \rangle \longrightarrow_D s_3} \quad \langle y := x + 1, s_3 \rangle \longrightarrow s_4}{\langle \text{begin var } x := 2; y := x + 1 \text{ end}, s_1 \rangle \longrightarrow s_2}$$

onde

|                                | $x$ | $y$ |
|--------------------------------|-----|-----|
| $s$                            | -   | -   |
| $s_0 = s[y \mapsto 1]$         | -   | 1   |
| $s_1 = s_0[x \mapsto 1]$       | 1   | 1   |
| $s_3 = s_1[x \mapsto 2]$       | 2   | 1   |
| $s_4 = s_3[y \mapsto 3]$       | 2   | 3   |
| $s_2 = s_4[\{x\} \mapsto s_1]$ | 1   | 3   |
| $s_5 = s_2[x \mapsto 4]$       | 4   | 3   |
| $s_6 = s_5[\{y\} \mapsto s]$   | 4   | -   |

**Exercício 4.2.** No exemplo anterior anota as árvores de derivação com as regras que são usadas em cada passo. Verifica que compreendes como são obtidos os estados/memórias  $s_2$  e  $s_6$ .  $\diamond$

## Referências

- [AM22] Torben AEGIDIUS Mogensén. *Programming Language Design and Implementation*. Texts in Computer Science. Springer International Publishing, 2022.
- [Fer04] Maribel Fernández. *Programming Languages and Operational Semantics*. King's College Publications, 2004.
- [NN07] H. Nielson and F. Nielson. *Semantics with Applications: an appetizer*. Springer, 2007.
- [Seb16] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 11th edition, 2016.